FAME
Software Management Plan

FAME Software Management Working Group

Version 2
June 13, 2000

Contents

Contents (continued)

1.0 Introduction

1.1 Purpose

The purpose of this document is to present a plan for managing the development of the FAME data analysis software. The plan addresses such issues as development methodology, standards, style, configuration control, and quality assurance. Essentially all of the ideas presented in this plan have been used in practice, and have been found to lead to software that is robust, maintainable, well documented, and portable to very high level.

1.2 Basis

The plan presented here is based on software development plans used successfully in two moderate-size software projects undertaken at USNO: MICA, the Multi-year Interactive Computer Almanac (Bangert and Kaplan 1992), and STELLA, the System to Estimate Latitude and Longitude Astronomically (Bangert 1996). The plan also draws on experiences gained on other astronomical software projects, especially the Sloan Digital Sky Survey (e.g. SDSS Collaboration 1996).

1.3 Software Management Options

The project to build the FAME data analysis software will almost certainly fall short of expectations, and quite possibly fail outright, without some level of management. One option is to implement formal software configuration management. However, the resources necessary to do this probably do not exist. Also, it is not desirable to create a management scheme that is so rigid that it limits creativity and flexibility. The software management plan presented here stresses a solid design, and practical rules and guidelines that are enforced by peer review.

1.4 Assumptions

1.4.1 Personnel

The most important recommendation regarding FAME software management — so important that it will be considered an assumption — is that the data analysis effort have a Software Project Manager and several dedicated pipeline programmers. These personnel will be fully dedicated to the development and maintenance of the operational pipeline software. The roles of these individuals, and other staff that will play critical roles in the software development effort, are defined below:

• *Algorithm developers*: Any member of the FAME team that contributes data reduction, file management, or other algorithms to the project.

• *Software Project Manager*: The individual responsible for ensuring that the operational pipeline software is of high quality, conforms to established standards, and is delivered on schedule and within budget. The software project manager also supervises the pipeline programmers.

• *Pipeline programmers*: FAME team members dedicated to development and maintenance of the operational data analysis software. Each pipeline programmer also serves as a *code reviewer* for code written by other pipeline programmers (see Section 7.1).

1.4.2 Other Assumptions

The scope of the software development task is described in Section 2.2.3 of the FAME Concept Study Report (Johnston 1999). The effort will likely be a complex, moderate-to-large size project that could take several years to complete. The software will have to be maintained through the lifetime of the space mission. Given the duration and nature of the task, there are several other assumptions that have been made in formulating the software development plan:

- Pipeline programmers may also be algorithm developers.
- The pipeline programmers will likely be scientists rather than trained software engineers.
- The pipeline programmers bring different skill levels and programming styles to the project.
- The pipeline programmers may change during the life of the project.
- Advances in computer hardware and operating systems may require that the software operate in an environment different from the one originally planned.
- Discovery of new, improved algorithms or techniques may require changes to the software (and perhaps to the design) in later stages of the software development.

2.0 Design Overview

This section presents some general concepts that will be useful in designing the FAME data analysis software system. The most basic design feature proposed here involves dividing the software system into two parts: the Executive and the Computational Engine.

2.1 Definitions

The *Executive* is the part of the software system that manages the data flow through the system at the highest level. The Executive also handles interactions with the user. Thus, the *User Interface* (UI), which queries the user for input and presents the user with output, is part of the Executive. The *Computational Engine* (CE) is the part of the system that performs the tasks requested by user input or by another part of the system.

2.2 Decouple the Executive and Computational Engine

Decoupling the computational part of the software system from the executive part is arguably the most important design goal for the software system. A generic example of this type of design is shown in Figure 1. It is relatively easy, in practice, to decouple the Executive and the CE. The highest-level modules in the CE serve as the interface between the Executive and the CE. These “task modules” take from the Executive the input necessary to perform a specific task, call lower-level modules within the CE that perform the task, then return the required output to the Executive. The output can be in the form of data arrays to be plotted, or even formatted tables to be displayed. It is the function of the Executive to display the results, or to pass the results to the next stop in the pipeline.

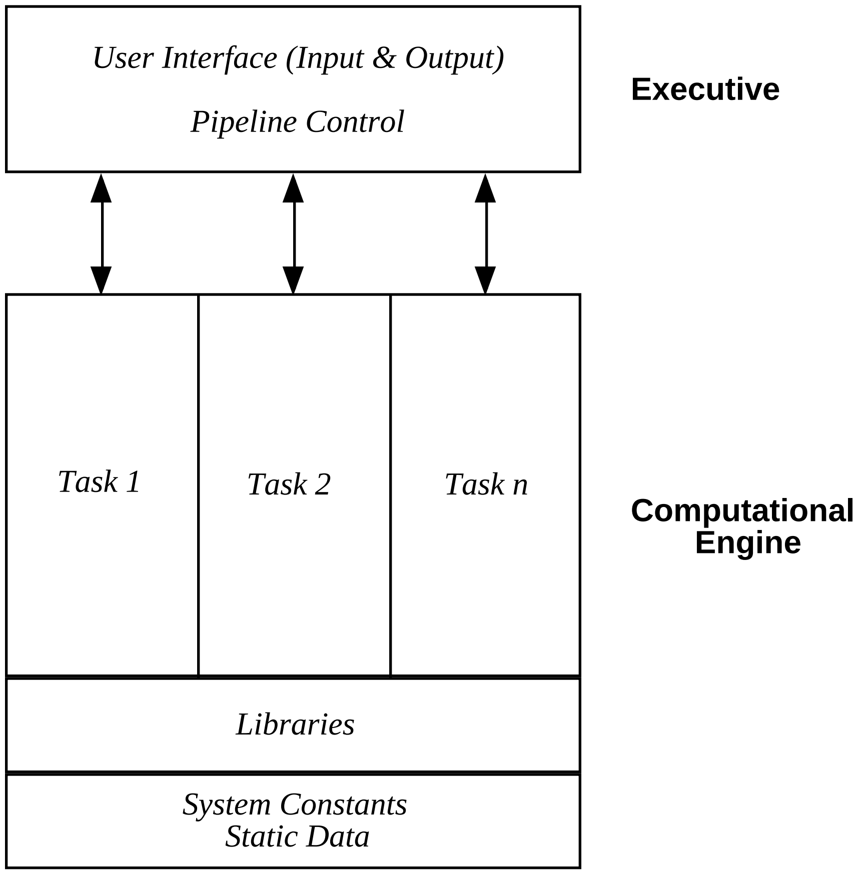

Figure 1: Schematic showing a software system design in which the Executive is decoupled from the Computational Engine (CE). The Executive controls the data flow though the system (pipeline) and is responsible for all interactions with the user (e.g obtaining input and displaying results). It may contain code that is platform-dependent. The CE performs the tasks requested by the user or other parts of the software system. In the case of FAME, all scientific data reduction functions (tasks) would reside in the CE. The CE is platform-independent, written in an ANSI-standard language of choice. The libraries contain logical blocks of code usually shared by several task modules. System constants are defined once, in a separate module, and available to all functions in the CE. Static data required by the system is handled the same way.

2.3 Platform Dependent and Platform Independent Code

The main reason for decoupling the executive code and the computational code is to isolate platform-dependent code—code that uses specific services available from the operating system or hardware. Platform-dependent code resides in the Executive, while the CE is platform-independent. The CE is made platform-independent by strict adherence to the ANSI standard for the programming language of choice. However, it may be necessary to use platform-dependent software, such as a GUI toolkit or an off-the-shelf graphics library, to create the data analysis system. This software is permitted within the Executive. The standards used in the Executive may be entirely different than those used in the CE. This general design creates a highly flexible, robust software system. If it becomes necessary to change platforms, the CE will port without modification. All that need be done is develop a new Executive, or modify the existing one, for the new platform.

2.4 Input/Output Operations

Most input/output (I/O) operations occur in the Executive, but there may be instances when proper design calls for an I/O operation in the CE. In these situations, only ANSI standard I/O is permitted. The operation should be isolated in a separate module within the CE and carefully documented.

2.5 Mixing Languages

Another advantage of decoupling the Executive and CE is that each can be written in the language best suited for the task to be performed. For example, GUIs are often written in C++ because numerous object-oriented toolkits for creating GUIs are readily available. In this example, it would make sense to write the Executive in C++. The CE, on the other hand, may best be written in one of the procedural languages used by scientists, such as C or Fortran. The binding of the two languages occurs in the Executive, at the boundary between the CE and the Executive. Languages should not be mixed within the CE itself.

2.6 Cross-Platform Development

A major advantage of making the CE platform-independent is that it can be developed on virtually any platform. Assume, for example, that Sun workstations running a specified version of the Solaris operating system have been selected for mission data analysis operations.   Certainly, the CE software must be thoroughly tested, and execute correctly, on this platform, but the software does not necessarily have to be developed on this platform. The CE software could be developed, for example, on inexpensive PCs, or any other platform that supports an ANSI compiler for the programming language of choice. Experience has shown that there are major advantages to this approach, as different compilers are sensitive to different coding errors or “dangerous” code constructions. Compiling and testing across different platforms helps to ensure a robust, portable CE.

2.7 Data Flow Diagrams

Data flow diagrams have already been used to model the FAME data analysis system (e.g. Johnston 1999; Reasenberg 1999). These diagrams identify processes that must take place, as well as the input and output (in a general form) to each process. The processes are identical to the tasks or subtasks that must be performed by the software. Thus, some form of data flow diagram is helpful in designing the software. Their use is encouraged.

3.0 Algorithms

Algorithms are procedures used to solve mathematical problems, or in this case, to perform the various tasks and sub-tasks required for FAME data analysis. Algorithms will be used by the pipeline programmers as the basis for the data analysis software. Thus, the algorithms represent a critical resource that must be managed.

3.1 Documentation

All algorithms accepted for use in the FAME data analysis software should be documented in both printed and electronic form. A FAME Technical Note series should be initiated and maintained to serve as a repository for the algorithms contributed to the project. The technical notes should be numbered in some logical form for easy reference. An index to the technical notes should exist in both printed and electronic form. The electronic form of the technical notes and index should be accessible to the FAME team via the FAME Web Site. Use of Portable Document Format (PDF) for the electronic version of the technical notes is recommended.

3.2 Numerical Examples

Every algorithm accepted for use in the FAME software should be accompanied by at least one numerical example including verified ("truth") results. The ability of the pipeline programmers to reproduce the example is the first step in unit-testing the software. Successful duplication of the numerical example gives the programmer an indication that the algorithm has been correctly implemented.

3.3 Prototype Code

Algorithm developers must test their algorithms before formally submitting them for use in the data analysis software. This will undoubtedly require programming the algorithm. Such code written during the development of an algorithm may not fully conform to FAME standards, and may not even be written in the official project language. This code—*prototype code*—is still valuable and should be managed (i.e. archived and subject to version control). The prototype code can be used by the pipeline programmer as the basis for the operational code to be used in the data analysis software. Furthermore, results from the actual code can be compared to results from the prototype code as a first check that the algorithm has been correctly implemented. Prototype code should be submitted with any associated test programs, and at least one test case including truth results.

4.0 Computational Engine

The CE is the part of the software system that performs the tasks requested by user input or by another part of the system. It is within the CE that all scientific data analysis computations should take place. The CE should be platform-independent: it should compile without errors, then execute identically to the required precision, on different platforms, without requiring modification. Platform independence is achieved by strict adherence to standards.

4.1 Language

The language selected for the computational engine must have an ANSI-standard set of features. It is also desirable to choose a language that is widely used in the scientific community. This narrows the selection to three languages: Fortran, C, and C++.

4.1.1 C++

C++ is a language specifically designed for object-oriented programming (OOP). Effective use of OOP requires knowledge of and considerable experience using object-oriented design (OOD). The advantages of OOD and OOP are well established. However, it will likely be difficult to assemble a team of scientific programmers with considerable experience in OOD, and this considerable experience is essential for minimizing the risk of a bad design and equally bad software. Thus, it is recommended that the language choices be limited to the procedural languages.

4.1.2 Fortran

The biggest advantages of Fortran are its good support for mathematics, its readability, and a well-established base of existing software libraries. On the negative side, Fortran—specifically the widely-used Fortran 77 and its predecessors—almost encourages poor programming practices due to the lack of some important control structures (such as the `while` loop), allowing multiple entry points and implicit typing, and many language extensions.

4.1.3 C

It is recommend that the CE be written in structured, ANSI C. Ritchie (1993) summarizes several key advantages of C. He notes that "… C has remained freer of proprietary extensions than other languages." C is also an extremely flexible language that is useful for a wide range of applications. As Ritchie states, "C is… efficient enough to replace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments." One disadvantage is that C programs tend to be obscure and difficult to read. This problem can be solved by effective use of coding standards.

Conforming to the ANSI C standard ensures that the CE will be highly portable regardless of platform. Setting appropriate compiler options for "ANSI strict" enforces compliance with the ANSI standard. The standard source for documenting ANSI C is Kernighan and Ritchie (1988). A good source of practical information on writing well-engineered ANSI C programs is Darnell and Margolis (1988).

4.2 Standards

The CE should be developed according to an established set of standards and guidelines. These standards ensure that the code is readable and has a consistent style, regardless of the programmer. The standards also ensure that the code is robust, maintainable, and meets minimum documentation requirements. The standards presented here are based on those used in the USNO projects cited in section 1.0, with additional contributions from the UK Starlink project (Charles and Murray 1994).

4.2.1 General Design Guidelines for Functions

See Section A.1 of Appendix A.

4.2.2 Specific Coding Rules and Guidelines

See Section A.2 of Appendix A.

4.2.3 Style Standards

See Section A.3 of Appendix A.

4.2.4 Error Handling Guidelines

See Section A.4 of Appendix A.

4.2.5 In-line Documentation Rules

See Section A.5 of Appendix A.

4.2.6 C Function Template

A proposed standard template for C functions is provided in Appendix B.

4.2.7 C Header File Template

A proposed standard template for C header (“`.h`”) files is provided in Appendix C.

4.2.8 Sample Code

Sample code illustrating the standards and guidelines cited in sections 4.2.1 through 4.2.5 is given in Appendix D.

4.3 Numerical Methods Library

It is recommended that a standard numerical methods library be adopted for use in the FAME data analysis software. This will provide a single source for commonly-used numerical functions, thus facilitating software maintenance. The library should be written in ANSI C, portable to all platforms involved in the data analysis. The actual library to be used is TBD.

5.0 The Executive

The Executive is the part of the software system that manages the data flow through the pipeline. It also solicits and accepts input from the user, and provides output to the user in forms such as tables and graphs. Thus, the user interface resides within the Executive. Any platform-dependent code required by the software system also resides in the Executive.

Given the high data rate and data volume associated with FAME, the data flow will be a very important issue. For example, in developing astrometric block-adjustment software, about 80% of the development time was spent in the data flow and file organization, and about 20% in the “math” coding (Zacharias 2000).

5.1 Language

There are obvious advantages to using the same language for the Executive as was used for the CE. However, this is not necessary. The goal is to use the language best suited for the actual tasks that the Executive must perform. The binding between the CE and the Executive should reside in the Executive at the boundary between the parts.

5.1.1 Candidates

· C: cross-platform; same language as the CE; ANSI standard.

- C++: cross-platform; standard. An advantage of using C++ as an executive language is that many GUI toolkits are available. Given the experience needed to effectively code in C++ (see Section 4.1.1), and the fact that there are no plans for a GUI in this project, C++ is probably not a good choice.
- Perl: see *http://www.p erl.com/pub/*; cross-platform; free. It is very different programming in Perl than in more familiar languages like C or Fortran, which is a strike against it (requiring additional learning curves should be well motivated).
- IDL: see *http://www.rsinc.com/idl/index.cfm*; somewhat cross-platform; expensive. IDL's strength is in its plotting abilities. However, there would be a large overhead associated with using IDL as the Executive language. It may not be ideal, or even able to manage the data flow.
- TCL/TK: see *http://dev.scriptics.com/software/*; cross-platform; free. TCL/TK was used successfully by SDSS. It does require additional programming (a C wrapper around every C function that must be available from the TCL command line), though its easy enough to copy templates for that. Its power is that it allows one to return data structures from C calls, and then pass them around at the command line level. The contents of the data structures can be plotted, examined, and passed to other TCL-wrapped C functions. TCL/TK has many desirable features for use as the Executive language.

5.1.2 Recommendation

The actual language to be used in the Executive is TBD. If a simple Executive is desired, C is probably the best choice. However, if a more powerful Executive is desired, one which pastes together computational functions with I/O, plotting, etc. in a command link interface, then TCL/TK is a good choice. The second choice will involve more effort.

5.2 Standards

It is difficult to adopt a specific set of standards for the Executive, given that the language is TBD and platform-dependent code is permitted. Furthermore, it may be necessary to use a third-party software package (e.g. a graphics package) within the Executive. This third-party software will naturally adhere to the standards set by its developers. Despite these difficulties, a few common sense guidelines can be stated:

- If the same language is used for both the Executive and the CE, then the standards adopted for the CE should be used in the Executive to the maximum extent possible.
- If the language chosen for the Executive has an ANSI standard, then the ANSI standard should be followed to the maximum extent possible.
- Platform dependencies and bindings to third-party software should be isolated in separate modules and especially well documented.
- The concepts behind many of the standards given in Appendix A can be applied to any programming language.

5.3 Plotting

The Executive should provide two plot capabilities. The first capability is to be able to plot data directly from the Executive, via calls to a standard plot library (TBD). The Executive should also provide a means to dump selected data to disk in various formats (FITS tables, ASCII files, etc.), allowing users the option of using their favorite standalone plotting packages.

6.0 Version Control

In a software development task of this size, an automated version control system (VCS) is essential. Simply put, a VCS records the history of critical project files. It allows a software developer to retrieve a past version of a document or source file. It also allows several developers to work on the same file simultaneously, merging the changes when the developers are done.

6.1 Strategy

Source code can be divided into *products*, where a *product* is a group of source code that is compiled together and has a single controlling “make” or project file. For example, each pipeline (centroid, sphere reconstruction, spiral reduction, photometry, etc) would be treated as a separate product. Each product generally is a different module for version control purposes. Each product is assigned a unique prefix, such as “ce” for centroiding or “ph” for photometry. All global functions and variables in a product must be preceded by its unique prefix. This avoids name collisions between products (this also applies to enumerated values).

6.2 Files Subject to Control

The following items should be subject to version control:

- operational source code
- calibration and parameter data files
- “make” or project files
- test programs
- test input data, “truth” results (if available), and actual test results
- prototype code (see section 3.3)
- code and algorithm documentation

6.3 Candidate VCS Products

The following products have been used in other USNO projects.

6.3.1 Microsoft Visual SourceSafe [*http://msdn.microsoft.com/ssafe/prodinfo/default.asp*]

Microsoft Visual SourceSafe (VSS) is being used successfully for the development of the next major version of STELLA (see section 1.0). Its primary advantage is integration with Microsoft development tools. The VSS server must run on a Windows 95 or later PC, although a Unix port is available from Mainsoft [*http://www.mainsoft.com/products/visual/over.html*]. A MacOS client is available from Metrowerks [*http://www.*

metrowerks.com/desktop/MWVSS/]. The VSS server can be accessed over the Internet using another third-party product (SourceOffSite; *http://www.sourcegear.com/SOS/*). VSS is quite expensive: each Windows client has a street price in the $300-$400 range. The server is included with the client, so it is essentially "free."

6.3.2 Concurrent Versions System [*http://www.gnu.org/software/cvs/cvs.html*]

Concurrent Versions System (CVS) is part of the GNU project, and thus is free software. CVS servers and clients run on most Unix variants [see *http://www.sourcegear.com/CVS/Dev/code*]. Clients for Windows [*http://www.wincvs.org/*] and MacOS [*http://www.maccvs.org/*] are also available. CVS supports access to its directories from remote hosts using standard Internet protocols. CVS has been used successfully in many projects, including the Sloan Digital Sky Survey, and is familiar to several USNO staff members. Due to its familiarity and low cost, CVS appears to be the better choice for development of the FAME data analysis software.

7.0 Quality Assurance

7.1 Code Reviews

Peer review of code proved to be a very effective way to enforce compliance with standards, to check the readability of the code, and to identify risky code constructions. Each function or subroutine should be read by another pipeline programmer, critiqued, then passed back to the original programmer for action. The software project manager arbitrates conflicts regarding suggested changes to the code.

7.2 Unit Testing

Each function or subroutine should be thoroughly tested as an individual unit using the tests provided by the algorithm developer. The programmer should also perform additional tests as appropriate. Emphasis should be placed on testing code response to special cases. Experience has shown that end-to-end testing of the system proceeds much more smoothly when the individual components have been thoroughly debugged first. It is usually much easier to debug a small block of code that performs a specific task, rather than postponing testing until a large block of code is done.

7.3 Subsystem Testing

Once the individual functions have been tested, the subsystems should be individually tested. For example, in Figure 1, the various "task" modules within the CE are the subsystems. There may be lower levels of subsystems (logical subsets of code within the task modules) depending upon the complexity of the code. Just as in unit testing, it is easier to test smaller blocks of code before testing the full system.

7.4 Regression Testing

The unit and subsystem tests should be executed by a top-level build on each software product. An error should be returned if any of the tests fail. All files necessary to conduct the tests must be included as part of the software product (that is, they are included in the code repository module for that product). Whenever a software product is modified, a top-level make should be executed to verify that the product both compiles and passes all of its regression tests before the modifications are checked in to the code repository.

7.5 Incident Tracking

Incident tracking software manages bug reports, suggestions (or requests) for change, and other "incidents" related to the operational software. Such software is very useful for the project manager, the programmers, and the software testers, as it provides a way to track the status and history of each issue. Use of incident tracking software in the FAME project is highly recommended. Two candidate systems are described below.

7.5.1 Candidate Incident Tracking Products

The following products have been used in other USNO projects.

7.5.1.1 TestTrack [*http://www.seapine.com/*]

This product was used successfully in the STELLA project. It is available in several versions providing a choice of platform-native or Web-based clients. Pricing starts at about $200-300, depending upon the version.

7.5.1.2 GNATS [*http://www.gnu.org/software/gnats/gnats.html*]

This product was used successfully in the SDSS project. GNATS is part of the GNU project, and thus is free software. A Web front-end for GNATS and a list of other GNATS resources are available via the links at *http://alumni.caltech.edu/~dank/gnats.html*. Due to its familiarity and low cost, GNATS appears to be the better choice for use in FAME software development.

8.0 Contributors

The FAME Software Management Working Group consisted of John Bangert (Chair), Greg Hennessy, Scott Horner, Jeff Munn, Marc Murison, and Bob Reasenberg. Ralph Gaume, as MO&DA Lead, served as an *ex officio* member of the group.

Nancy Oliversen and Norbert Zacharias also contributed useful comments on the first draft of the plan.

9.0 References

Bangert, J. and Kaplan, G. (1992). *BAAS* **24**, 740.

Bangert, J. (1996). *Chips*, Vol. XIV, Issue 2, p. 5 (April 1996).

Charles, A. and Murray, J. (1994). “Starlink C Programming Standard.” (Rutherford Appleton Laboratory: Starlink General Paper 4.2).

Johnston, K. (1999). FAME Concept Study Report (NASA Midex AO-98-OSS-03).

Darnell, P. and Margolis, P. (1988). *C: A Software Engineering Approach*. (Springer-Verlag).

Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. (Prentice Hall).

Reasenberg, R. (1999). “An Approach to FAME Data Processing.” (internal FAME project document presented at USNO, July 1999).

Ritchie, D. (1993). “The Development of the C Language,” paper presented at the Second History of Programming Langauges conference; Cambridge, MA; April 1993.

Schildt, H. (1997). *C/C++ Programmer’s Reference*. (Osborne).

SDSS Collaboration. 1996, SDSS Black Book (Princeton: Dept. Astron., Princeton Univ.). See also *http://www.astro.princeton.edu/PBOOK/welcome.htm*.

Zacharias, N. (2000); private communication.

Appendix A: C Coding Standards

A.1 General Function Design Guidelines

- each C function should perform a single, specific task
- strive for readable, understandable code over elegant-but-incomprehensible code
- avoid the more obscure parts of the C language
- each function should be designed as generally as possible; code should handle all relevant "special cases"
- each function should be designed using a simple, top-down flow (no back transfers except for loops)
- unused variables or unreachable code are not permitted in operational software

A.2 Specific Rules and Guidelines

Header File

- each file of C source must be accompanied by an associated standard header file of the same name except with a `.h` extension.
- the format of this header file is given in Appendix C.
- the header file must contain, as applicable:
  - list of external variables used in the associated source file (`extern`)
  - `#include` statements
    - external header files
    - standard libraries
  - data structures
  - function prototypes

Data structures

- data structures (derived data types) are to be defined in header files using the `typedef` keyword. For example, the structure `body` is defined by:

```
typedef struct
   {
      short int type;
      short int number;
      char name[100];
   } body;
```

Use of macros

- it is recommended that use of macros be avoided

C keywords

- do not redefine names of C keywords

Use of global variables

- minimize use of global variables, but…
- define physical, mathematical, and system constants *once* in a separate function; pass to all modules that require them via global variables

Input/output (I/O)

- isolate all I/O; place all I/O operations in separate functions; do not mix I/O operations with computations

Function and variable typing

- functions must be explicitly typed, including `void`
- `void` must be explicitly stated in an empty argument list
- implicit type conversions (type promotions) are not permitted; use explicit type casting
- `char` variables are not to be used as `int` variables
- variables of type `int` must be declared as `long` or `short`

Operations in argument lists

- avoid operations in argument lists

Use of `goto`

- `goto` statements are prohibited

Numerical values in an `if` *expression*

- use of numeric values in an `if` expression is permissible, but should be commented

Incrementing/decrementing variables

- avoid incrementing/decrementing variables via operators in an assignment statement (e.g. `j = i++;`)
- avoid incrementing or decrementing pointer variables

`switch` statement

- the last `case` in a `switch` statement should end with a `break` statement even though it is not required

Loops

- avoid changing the current loop index and range within a `for` loop
- use only non-floating loop variables for the index and range

Use of `sizeof`

- for clarity and portability, use the `sizeof` function explicity wherever appropriate

Preprocessor commands

- limit use of preprocessor commands; confine to header files (except for conditionals and `#include`)

A.3 Style

General

- each line of source code is limited to 80 characters, but continuations are permitted
- no tab characters are permitted in source code files; use spaces instead
- only one C statement per line of code is permitted

Use of case

- all local variables are to be lowercase
- all global variables are to be UPPERCASE

Use of Spaces

- function argument list: use space after each comma; a blank line must appear between input and output arguments
- use a space after the name of each called function; no space should appear between an array name and the opening square bracket
- use a space after a C keyword
- use a space after each semicolon in a `for` statement
- a single white space is to be used on each side of an equal sign and on each side of an operator, including relational operators
- do not include whitespace inside compound operators such as `+=`, `-=`, etc.

Curly Braces & Indenting

- all curly braces must appear on their own lines
- code following an opening curly brace must be indented three spaces; code reverts to original column after a closing curly brace
- opening and closing curly braces (corresponding curly braces) must appear in the same column
- curly braces used in an `if` statement should appear under the `i` in `if`; the `else` should be indented one space

Argument lists

- all input variables should be given first, followed by a blank line, followed by output variables
- avoid variables that are input, changed withinh the function, then output.

A.4 Error Handling Guidelines

- all input values should undergo “sanity checks” at the highest level of the structure
- the program should provide basic error checks on all relevant and important internal calculations

- code should always provide a “graceful exit,” not a crash, when an anomaly or error is encountered

- when a function determines and sets an error or status flag, the `return` value from the function should be the flag

A.5 In-line Documentation Rules

- each file of C source code is to be preceded with a comment block as follows:

```
/*
   source file name: description

   Version: x.x
*/
```

- comment delimiters (`/*` and `*/`) are to appear on their own lines, in the leftmost columns (except in a “function delimiter”; see below)

- the amount of documentation (comments) within a function is adequate if the comments are able to “stand alone” in explaining how the function works

- text of comments is to appear in the lines between the comment delimiters, and must be indented so as to be aligned with the code

- no comments are to appear on the same line as code

- make liberal use of blank lines to make code more readable

- each routine must be preceded by a “function delimiter”: a comment line in which the `/*` is typed in columns 1 and 2, columns 3-9 are padded with asterisks, and the function name is typed beginning in column 10, followed by a space and a `*/`

- each function must include a standard prolog which provides the following information: purpose, references, input/output arguments (including types and dimensions), return value, global variables used, functions called, version/date/programmer, and notes; see Appendix B

Appendix B: C Function Template

```
/********function_name */

(function name and argument list)
/*
-------------------------------------------------------------------------

   PURPOSE:
      x

   REFERENCES:
      x

   INPUT
   ARGUMENTS:
      x

   OUTPUT
   ARGUMENTS:
      x

   RETURNED
   VALUE:
      x

   GLOBALS
   USED:
      x

   FUNCTIONS
   CALLED:
      x

   VER./DATE/
   PROGRAMMER:
      V1.0/mm-yy/initials (organization)

   NOTES:
      x

-------------------------------------------------------------------------
*/
{
   short int ;

   long int ;

   float ;

   double ;

   (derived types)

   (code)

   return;
}
```

Note:

- Input/output arguments:
  - first line: indent three spaces, must contain variable name and data type;
  - following lines: indent three spaces, describe the variable;

  Example:

```
INPUT
ARGUMENTS:
   var_name[10] (short int; array)
      This variable is used as an example in this sample prolog.
   dummy (double)
      This is another example.
```

Appendix C: C Header File Template

```
/*
   Header file for source-file name
   Version x.x
/*

#ifndef MACRO-NAME__
   #define MACRO-NAME__

/*
   External variables
*/
   extern ...;

/*
   Standard Libraries
*/

   #include <standard library.h>
.
     .
     .
/*
   External files
/*

   #include "external file.h"
     .
     .
     .

/*
   Function prototypes
*/
     .
     .
     .

#endif
```

- *MACRO-NAME* is the name of the associated C source file, in upper case
- version number must match the version number of the associated C source file
- the ‘extern’ type modifier is used to specify all variables defined outside of the associated C source file
- standard libraries include `math.h`, `stdio.h`, etc.
- the header files for files containing functions used within *MACRO-NAME*.c are to be included under “External files”
- function prototypes should be in the form used in the associated C source file
- At the top of the associated file of C source, but after the comment block, the following should appear:

```
#ifndef _MACRO-NAME_
   #include "macro-name.h"
#endif
```

Appendix D: Sample Code

```
/********precession */

void precession (double tjd1, double *pos, double tjd2,

                 double *pos2)
/*
------------------------------------------------------------------------

   PURPOSE:
      Precesses equatorial rectangular coordinates from one epoch to
      another.  The coordinates are referred to the mean equator and
      equinox of the two respective epochs.

   REFERENCES:
      Explanatory Supplement to AE and AENA (1961); pp. 30-34.
      Lieske, J., et al. (1977). Astron. & Astrophys. 58, 1-16.
      Lieske, J. (1979). Astron. & Astrophys. 73, 282-284.
      Kaplan, G. H. et. al. (1989). Astron. Journ. Vol. 97,
         pp. 1197-1210.
      Kaplan, G. H. "NOVAS: Naval Observatory Vector Astrometry
         Subroutines"; USNO internal document dated 20 Oct 1988;
         revised 15 Mar 1990.

   INPUT
   ARGUMENTS:
      tjd1 (double)
         TDB Julian date of first epoch.
      pos[3] (double)
         Position vector, geocentric equatorial rectangular coordinates,
         referred to mean equator and equinox of first epoch.
      tjd2 (double)
         TDB Julian date of second epoch.

   OUTPUT
   ARGUMENTS:
      pos2[3] (double)
         Position vector, geocentric equatorial rectangular coordinates,
         referred to mean equator and equinox of second epoch.

   RETURNED
   VALUE:
      None.

   GLOBALS
   USED:
      T0, RAD2SEC

   FUNCTIONS
   CALLED:
      None.

   VER./DATE/
   PROGRAMMER:
      V1.0/01-93/TKB (USNO/NRL Optical Interfer.) Translate Fortran.
      V1.1/08-93/WTH (USNO/AA) Update to C Standards.
      V1.2/03-98/JAB (USNO/AA) Change function type from 'short int' to
                               'void'.
      V1.3/12-99/JAB (USNO/AA) Precompute trig terms for greater
                               efficiency.

   NOTES:
      1. This function is the "C" version of Fortran NOVAS routine
      'preces'.

------------------------------------------------------------------------
*/
{
   double xx, yx, zx, xy, yy, zy, xz, yz, zz, t, t1, t02, t2, t3,
      zeta0, zee, theta, cz0, sz0, ct, st, cz, sz;

/*
   Compute time arguments.  't' and 't1' correspond to Lieske's "big T"
   and "little t".
*/

   t = (tjd1 - T0) / 36525.0;
   t1 = (tjd2 - tjd1) / 36525.0;
   t02 = t * t;
   t2 = t1 * t1;
   t3 = t2 * t1;

/*
   Compute precessional angles in arcseconds.  'zeta0', 'zee', 'theta'
   below correspond to Lieske's "zeta-sub-a", "z-sub-a", and
   "theta-sub-a".
*/

   zeta0 = (2306.2181 + 1.39656 * t - 0.000139 * t02) * t1
         + (0.30188 - 0.000344 * t) * t2 + 0.017998 * t3;

   zee = (2306.2181 + 1.39656 * t - 0.000139 * t02) * t1
       + (1.09468 + 0.000066 * t) * t2 + 0.018203 * t3;

   theta = (2004.3109 - 0.85330 * t - 0.000217 * t02) * t1
         + (-0.42665 - 0.000217 * t) * t2 - 0.041833 * t3;

/*
   Convert the angles to radians.
*/

   zeta0 /= RAD2SEC;
   zee /= RAD2SEC;
   theta /= RAD2SEC;

/*
   Precalculate trig terms.
*/

   cz0 = cos (zeta0);
   sz0 = sin (zeta0);
   ct = cos (theta);
   st = sin (theta);
   cz = cos (zee);
   sz = sin (zee);

/*
   Compute the elements of the precession rotation matrix.
*/

   xx =  cz0 * ct * cz - sz0 * sz;
   yx = -sz0 * ct * cz - cz0 * sz;
   zx = -st * cz;
   xy = cz0 * ct * sz + sz0 * cz;
   yy = -sz0 * ct * sz + cz0 * cz;
   zy = -st * sz;
   xz = cz0 * st;
   yz = -sz0 * st;
   zz = ct;

/*
   Perform rotation.
*/

   pos2[0] = xx * pos[0] + yx * pos[1] + zx * pos[2];
   pos2[1] = xy * pos[0] + yy * pos[1] + zy * pos[2];
   pos2[2] = xz * pos[0] + yz * pos[1] + zz * pos[2];

   return;
}

/********set_body */

short int set_body (short int type, short int number, char *name,

                    body *cel_obj)
/*
------------------------------------------------------------------------

   PURPOSE:
      Sets up a structure of type 'body' - defining a celestial object-
      based on the input parameters.

   REFERENCES:
      None.

   INPUT
   ARGUMENTS:
      type (short int)
         Type of body
            = 0 ... major planet, Sun, or Moon
            = 1 ... minor planet
      number (short int)
         Body number
            For 'type' = 0: Mercury = 1,...,Pluto = 9, Sun = 10,
                            Moon = 11
            For 'type' = 1: minor planet number
      *name (char)
         Name of the body.

   OUTPUT
   ARGUMENTS:
      struct body *cel_obj
         Structure containg the body definition (defined in novas.h)

   RETURNED
   VALUE:
      (short int)
         = 0 ... everything OK
         = 1 ... invalid value of 'type'
         = 2 ... 'number' out of range

   GLOBALS
   USED:
      None.

   FUNCTIONS
   CALLED:
      None.

   VER./DATE/
   PROGRAMMER:
      V1.0/06-97/JAB (USNO/AA)
      V1.1/10-98/JAB (USNO/AA): Change body name to mixed case.

   NOTES:
      None.

------------------------------------------------------------------------
*/

{
   char temp;

   short int error = 0;
   short int i;

/*
   Initialize the structure of type ‘body’.
*/

   cel_obj->type = 0;
   cel_obj->number = 0;
   strcpy (cel_obj->name, "  ");

/*
   Set the body type.
*/

   if ((type < 0) || (type > 1))
      return (error = 1);
    else
      cel_obj->type = type;

/*
   Set the body number.
*/

   if (type == 0)
      if ((number <= 0) || (number > 11))
         return (error = 2);
    else
      if (number <= 0)
         return (error = 2);

   cel_obj->number = number;

/*
   Set the body name in mixed case.
*/

    i = 0;
    while (name[i] != 0)
    {
       if (i == 0)
          temp = toupper (name[i]);
        else
          temp = tolower (name[i]);
       cel_obj->name[i++] = temp;
    }
    cel_obj->name[i] = '\0';

    return (error);
}
```